A general lock-free algorithm using compare-and-swap

Gao, H.; Hesselink, W.H.

Published in:
Information and Computation

DOI:
10.1016/j.ic.2006.10.003

Available online at www.sciencedirect.com
ScienceDirect
Information and Computation 205 (2007) 225–241
Information and Computation
www.elsevier.com/locate/ic

A general lock-free algorithm using compare-and-swap

H. Gao [a], W.H. Hesselink [b,*]

[a] *School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China*

[b] *Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands*

Received 28 October 2004; revised 28 June 2006
Available online 14 December 2006

Abstract

The compare-and-swap register (*CAS*) is a synchronization primitive for lock-free algorithms. Most uses of it, however, suffer from the so-called ABA problem. The simplest and most efficient solution to the ABA problem is to include a tag with the memory location such that the tag is incremented with each update of the target location. This solution, however, is theoretically unsound and has limited applicability. This paper presents a general lock-free pattern that is based on the synchronization primitive *CAS* without causing ABA problem or problems with wrap around. It can be used to provide lock-free functionality for any data type. Our algorithm is a *CAS* variation of Herlihy's *LL*/*SC* methodology for lock-free transformation. The basis of our techniques is to poll different locations on reading and writing objects in such a way that the consistency of an object can be checked by its location instead of its tag. It consists of simple code that can be easily implemented using C-like languages. A true problem of lock-free algorithms is that they are hard to design correctly, which even holds for apparently straightforward algorithms. We therefore develop a reduction theorem that enables us to reason about the general lock-free algorithm to be designed on a higher level than the synchronization primitives. The reduction theorem is based on Lamport's refinement mappings, and has been verified with the higher-order interactive theorem prover PVS. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and formulate without considering the internal structure of the final implementation.

* Corresponding author. Fax: +31503633800.
E-mail addresses: huigao@uestc.edu.cn (H. Gao), w.h.hesselink@rug.nl (W.H. Hesselink).

0890-5401/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2006.10.003

Keywords: Lock-free; Refinement mapping; Atomicity; Compare-and-swap; Correctness

1. Introduction

We are interested in designing efficient data structures and algorithms on shared-memory multiprocessors. A natural model for these machines is an asynchronous parallel machine, in which the processes may execute instructions at a different rate, and are subject to long delays. On such machines, processes often need to coordinate with each other via shared data structures. In order to prevent the corruption of these concurrent objects, processes need a mechanism for synchronizing their access. The traditional approach is to explicitly synchronize access to shared data by different processes to ensure correct behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

If the blocked process is performing a high-priority or real-time task, it is highly undesirable to halt its progress. Due to blocking, the classical synchronization paradigms using locks can incur many problems such as long delays, convoying, priority inversion and deadlock. Using locks also involves a trade-off between coarse-grained locking which can significantly reduce opportunities for parallelism, and fine-grained locking which requires more careful design and is more prone to bugs.

This state of affairs has led to the search for lock-free concurrent data structures [10]. A concurrent data structure is *lock-free* [10,6] if it guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. Indeed, lock-free data structures are immune from the aforementioned problems. In addition, they can offer progress guarantees, and increase performance by allowing extra concurrency.

Herlihy [10] has shown that the *compare-and-swap* (*CAS*) primitive and the similar *load-linked* (*LL*)/*store-conditional* (*SC*) are universal primitives that solve the consensus problem. A number of researchers, e.g. [4,5,11,12,18,20], have proposed techniques for designing *lock-free* concurrent data structures. The basis of these techniques is using some synchronization primitives such as *CAS* or *LL*/*SC*.

Many machines provide either *CAS* or *LL*/*SC*, but not both. All architectures that support *LL*/*SC* restrict memory accesses between *LL* and *SC*. Furthermore, most kinds of hardware do not provide the complete semantics of *LL*/*SC* that might be expected by some program designers. For example, the cache-coherence mechanism may let *SC* fail spuriously, i.e., a *SC* operation may fail if a cached word is selected for replacement by the cache protocol. Some machines such as DEC Alpha and PowerPC, also restrict *LL*/*SC* operations from being concurrently executed since *LL* and *SC* are implemented using only one tag bit per processor.

The *CAS* operation takes the address of a memory location, an expected value, and a new value. If the location contains the expected value, the *CAS* operations atomically stores the new value in the location and returns *true*. Otherwise, the contents of the location remain unchanged,

and the *CAS* returns *false*. The *CAS* is said to succeed when it returns *true*, and to fail when it returns *false*.

Associated with most uses of *CAS* (and restricted *LL*/*SC*) is the ABA problem [15], which can be described as follows [6]. A typical way to use *CAS* is to read a value —call it A— from a location, and to then use *CAS* to attempt to change the location from A to a new value. The intent is often to ensure that the *CAS* succeeds only if the location's value does not change between the read and the *CAS*. However, the location might change to a different value B and then back to A again between the read and the *CAS*, in which case the *CAS* succeeds. This phenomenon is known as the ABA problem and is a common source of bugs in *CAS*-based algorithms.

The simplest and most efficient solution to the ABA problem is to include a tag with the memory location such that the tag is incremented with each update of the target location [24]. This solution with tags in principle requires that the tags are unbounded. The practical solution of taking 32-bit integers for the tags gives an infinitesimal but positive probability of misbehaviour by wrap around. In practice, this solution worked for 32-bit architectures that supported double-word *CAS*. According to [6], the technique cannot be used in the emerging 64-bit architectures that only support 64-bit *CAS*. Our present algorithm does not have this problem.

1.1. Using CAS for atomic read-and-modify

In this paper, we develop a reduction theorem, Theorem 4.1, that enables us to reason about a lock-free program on a higher level than the synchronization primitives. The algorithm enables us to atomically inspect and modify the contents of a system of N variables of the same type, by means of $K \geqslant N + 2P$ implemented variables and $N + K$ *CAS* registers with values bounded by K. Here, P is the number of processes. One can take $K = N + 2P$, but bigger K will give better performance.

Our algorithm is a generalization of Herlihy's general methodology for lock-free transformation [11]. The basis of our techniques is to poll different locations on reading and writing objects, in such a way that the consistency of an object can be checked by its location instead of its tag. It consists of simple code that can be easily implemented using C-like languages.

Theorem 4.1 is based on refinement mappings as described by Lamport [17], which are used to prove that a lower-level specification correctly implements a higher-level one. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and formulate, without considering the internal structure of the final implementation. In particular, nested loops in the algorithm may be eliminated at a time.

In [9], we have shown a similar reduction theorem for reducing lock-free implementations using *LL*/*SC*. This time, we aim to provide a correct lock-free transformation using *CAS*.

1.2. Related work

The field of lock-free algorithms and data structures is very active and rapidly growing. Here, we only mention some recent contributions.

One can distinguish three levels, but several of the papers to be listed serve on more than one level. On the highest level there are lock-free implementations of memory management systems [23] and garbage collectors [8], and special data structures like queues [22], linked lists, and hash tables [7,21,25].

There is an intermediate level where the primitives *CAS* or *LL/SC* are used to build atomic abstractions that can be used in the constructions of the higher levels. Our present algorithm fits into this level since it enables us to concurrently inspect and modify the contents of a system of variables and its specification was designed for use in the garbage collector of [8]. This is also the level of Michael's safe memory reclamation technique [22], and of partial memory management algorithms like in [12,14].

On the lowest level, there are implementations of *LL/SC* by means of *CAS* or restricted forms of *LL/SC*. In combination with our *LL/SC*-based algorithm of [9], these could be used to construct solutions to our present specification. As far as we know, the first lock-free implementation of *LL/SC* from *CAS* are in [2,24]. Both algorithms only implement small *LL/SC* objects. The first practical implementations of *LL/SC* using *CAS* are due to Jayanti and Petrovic [16]. The space requirements of their implementations do not scale well when the number of *LL/SC* variables is large. In order to implement N *LL/SC* variables, their algorithms require $\mathcal{O}(N)$ local variables per process. This amounts to $\mathcal{O}(NP)$ space for a system of P processes, in contrast to $\mathcal{O}(N+P)$ space that, e.g., our algorithm requires. To improve on Jayanti and Petrovic's implementations, Doherty et al. [6] and Michael [23] present lock-free algorithms that use $\mathcal{O}(N+P)$ space and $\mathcal{O}(N)$ space, respectively. Since the algorithms of [2,6,12,23,24] employ version numbers to eliminate the ABA problem, their theoretical correctness depends on unbounded version numbers.

2. Preliminaries

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space. There can be several processes running on a single processor. Let us assume there are P ($\geqslant 1$) concurrently executing sequential processes.

The processes communicate by reading and writing shared variables. All processes have their own private variables. A collection of values of all variables, shared and private, is called a *state* of the system. The set of all states is called the *state space* and is denoted Σ. If $\mathcal{C}$ is a command, we write $\mathcal{C}.p$ to denote the transition relation between states that corresponds to execution of command $\mathcal{C}$ by process p. So $(s,t) \in \mathcal{C}.p$ indicates that in state s process p can do a step $\mathcal{C}$ that establishes state t. When discussing the effect of a transition $\mathcal{C}.p$ from state s to state t on a variable v, we write v for the value of v in state s and v' for the value of v in state t. The union of the transition relations of all commands for process p is the *step* relation of p, denoted by $\mathcal{N}.p$.

Not all variables are equally relevant for the specification. The specification formalism therefore includes an *observable state space* Σ_0, and allows us to specify an *observation function* $\Pi : \Sigma \to \Sigma_0$. Usually, Σ_0 is spanned by a selection of the variables spanning Σ, and Π is a restriction of the state to these so-called visible variables. We assume that all levels of specifications under consideration have the same observable state space Σ_0.

We use the convention that shared variables are written in type writer font and the private variables are slanted. Outside of the programs, a private variable v of process p is denoted by $v.p$.

2.1. Specifications

We define a *specification* $\mathcal{S}$ to be a five-tuple $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$ where Σ is the state space, Π is the observation function, Θ is the predicate that indicates the initial states, $\mathcal{N}$ is the next state relation, and $\mathcal{L}$ is the supplementary property of the system (i.e., a predicate on infinite sequences of states to express liveness properties). It follows that the triple $(\Sigma, \Theta, \mathcal{N})$ is a *transition system* [19].

The next-state relation $\mathcal{N}$ is supposed to be reflexive. In this way, stuttering steps are allowed, i.e., steps in which the state does not change. These may represent invisible internal steps. In the presence of several processes as above, we have $\mathcal{N} = id \cup \bigcup_p \mathcal{N}.p$ where id is the identity relation.

The supplementary property $\mathcal{L}$ is a liveness property to ensure that eventually something good happens. Such a property is needed since the transition system only specifies safety requirements and has no kind of fairness conditions or liveness assumptions built into it.

An infinite sequence of states, say τ, is defined to be an *execution* of specification $\mathcal{S}$ if it satisfies the initial predicate Θ and the next-state relation $\mathcal{N}$, i.e., $\Theta(\tau_0)$ holds and $(\tau_n, \tau_{n+1}) \in \mathcal{N}$ for all n. We define a *behavior* of $\mathcal{S}$ to be an execution that also satisfies the supplementary property $\mathcal{L}$. The *visible behaviors* of $\mathcal{S}$ are the infinite sequences obtained by applying Π to the behaviors of $\mathcal{S}$.

A specification $\mathcal{S}_c$ is defined to *implement* a specification $\mathcal{S}_a$ if every visible behavior of $\mathcal{S}_c$ is also a visible behavior of $\mathcal{S}_a$, possibly after adding stutterings to it [1]. In this situation, $\mathcal{S}_c$ is regarded as the concrete specification and $\mathcal{S}_a$ as the abstract one.

When arguing about the correctness of programs and algorithms, it is preferable to use so-called assertional methods that reduce the investigations from behaviors to states and the next state relation as much as possible. Therefore, refinement mappings are introduced to prove implementation relations.

2.2. Refinement mappings

Let $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c, \mathcal{L}_c)$ and $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a, \mathcal{L}_a)$ be specifications, where $\mathcal{S}_c$ is regarded as concrete and $\mathcal{S}_a$ as abstract. A function $\varphi : \Sigma_c \to \Sigma_a$ is defined to be a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, notation $\varphi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$, if it satisfies:

(1) function φ preserves the observations: $\Pi_a(\varphi(s)) = \Pi_c(s)$ for every $s \in \Sigma_c$.
(2) function φ maps initial states into initial states: $\Theta_c(s) \;\Rightarrow\; \Theta_a(\varphi(s))$ for every $s \in \Sigma_c$.
(3) function φ preserves the next state relation: there is an invariant Q on Σ_c such that, for every pair of states $(s, t) \in \mathcal{N}_c$ with $Q(s)$, it holds that $(\varphi(s), \varphi(t)) \in \mathcal{N}_a$.
(4) function φ maps behaviors of $\mathcal{S}_c$ into behaviors of $\mathcal{S}_a$.

In our application below, we indeed need an invariant Q as allowed in condition 3. The following theorem of [1] is well-known and easy to prove.

Theorem 2.1. *If there exists a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$, then $\mathcal{S}_c$ implements $\mathcal{S}_a$.*

Refinement mappings give us the ability to reduce an implementation by reducing its components in relative isolation, and then gluing the *reductions* together with the same structure as the implementation. Atomicity guarantees that a parallel execution of a program gives the same results as a sequential and nondeterministic execution. This allows us to use the refinement calculus for

stepwise refinement of transition systems [3]. Essentially, the reduction theorem allows us to design and verify the program on a higher level of abstraction. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level.

The refinement relation is transitive, which means that we do not have to reduce the implementation in one step, but can proceed from the implementation to the specification through a series of smaller steps.

2.3. Synchronization primitives

Traditional multiprocessor architectures have included hardware support only for low-level synchronization primitives such as *CAS* and *LL*/*SC*, while high-level synchronization primitives such as locks, barriers, and condition variables have to be implemented in software.

CAS atomically compares the contents of a location with a value and, if they match, stores a new value at the location. The semantics of *CAS* is given by equivalent atomic statements below.

```
proc CAS(ref x; in old, new) : bool =
    ⟨  if x = old then x := new ; return true
       else return false fi ⟩
```

LL and *SC* are a pair of instructions, closely related to the *CAS*, and together implement an atomic Read/Write cycle. Instruction *LL* first reads the content of a memory location, say x, and marks it as "reserved" (not "locked"). If no other processor changes the content of x in between, the subsequent *SC* operation of the same process succeeds and modifies the value stored; otherwise it fails. There is also a validate instruction *VL* to check whether x was not modified since the corresponding *LL* instruction was executed. For the semantics of *LL*, *SC* and *VL*, we refer the interested reader to [9].

An atomic counter can be implemented by *fetch-and-increment* (*FAI*) and *fetch-and-decrement* (*FAD*) given below. Both operations return the original value of a memory location after atomically increment and decrement the counter, respectively. From hardware point of view, they are simpler versions of *CAS*.

```
proc FAI(ref x) : int =
    ⟨  x := x + 1; return x − 1; ⟩
```

FAD is declared analogously. When *FAI* and *FAD* are not available on the machine architecture, they can be easily implemented by *CAS* and *LL*/*SC*. For example, *FAI* can be implemented by *CAS* in the following lock-free way:

```
proc FAI(ref x) : int =
local y ;
    loop
      y := x ;
      if CAS(x, y, y + 1) then return y fi
    end
end
```

3. The lock-free pattern

At the cost of copying an object's data before an operation, Herlihy [11] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *LL* and *SC*. Below in section 3.1, we describe this methodology and its formalization as done in [9].

This paper is devoted to an implementation of this interface by means of *CAS*. The interface is given in Fig. 1. There are P processes, concurrently involved in inspecting and modifying the contents of N shared nodes of type `nodeType`, interleaved with other activity.

For each process, this is modelled by means of an infinite loop in which the process alternates between a_1 and a_2. At a_1, process p does some noncritical activity on a shared variable `pub` and its own private variables $priv.p$ and $tm.p$, and determines an index $x.p$ for a node to modify in the next step. At a_2, it conditionally modifies `Node`$[x.p]$ based on the value of $priv.p$, which may yield a result $tm.p$. The action at a_2 is enclosed by angular brackets $\langle \ldots \rangle$ to indicate that it is defined as atomic. More precisely, we use

(1) *noncrit*(**ref** `pub` : `aType`, *priv* : `bType`; **in** *tm* : `cType`; **out** $x : [1 \ldots N]$) represents an atomic noncritical action on variables *pub* and *priv* according to the value of *tm*, and chooses an index x of a shared node to be accessed.
(2) *guard*(**in** `nd` : `nodeType`, *priv* : `bType`), a nonatomic boolean test on the variable `nd` of `node Type`. It may depend on private variable *priv*.
(3) *com*(**ref** `nd` : `nodeType`; **in** *priv* : `bType`; **out** *tm* : `cType`) : a nonatomic action on the variable `nd` of `nodeType` with result parameter *tm*. It is allowed to inspect private variable *priv*.

Constant
  $P =$ `number of processes`;
  $N =$ `number of nodes`;
Shared variable
  `pub`: `aType`;
  `Node`: **array** $[1 \ldots N]$ **of** `nodeType`;
  `opc` : `natural`; % auxiliary
Private variable
  priv: `bType`; *pc*: $\{a_1, a_2\}$; x: $[1 \ldots N]$; *tm*: `cType`;

Program
  loop
a_1: *noncrit*(`pub`, *priv*, *tm*, x);
a_2: $\langle$ **if** *guard*(`Node`$[x]$, *priv*) **then** *com*(`Node`$[x]$, *priv*, *tm*); **fi**; `opc ++` $\rangle$
  end.

Initial condition
  Θ_a: $\forall p \in [1 \ldots P] : pc.p = a_1$
Liveness
  $\mathcal{L}_a$: $\forall p \in [1 \ldots P], n \in \mathbb{N} : \Box(\texttt{opc} = n \ \wedge \ pc.p \neq a_1 \ \Rightarrow \ \Diamond(\texttt{opc} > n))$

Fig. 1. Interface $\mathcal{S}_a$.

The private variable *tm* is separated from *priv* since this enables us to assume that command *com* does not modify *priv* and yields *tm* as a result. Notice that *guard* and *com* are specified as non-atomic, but that command a_2 is specified as atomic. The atomicity must be guaranteed by the implementation.

In specification $\mathcal{S}_a$, lock-freedom is expressed by means of an auxiliary shared variable `opc` that counts the number of completed operations. This variable is therefore incremented in a_2. Lock-freedom now means that, whenever some process is in an operation, i.e., not at a_1, then eventually `opc` increases.

In the lock-free pattern, we are not interested in the internal details of these schematic commands but in their behavior with respect to lock-freedom.

In both implementations, we use a nonatomic read operation

read(**ref** `nd : nodeType`, **in** `nv : nodeType`),

that reads the value from `nv` into the variable `nd`. If `nv` is modified during *read*, the resulting value of `nd` is unspecified but type correct, and no error occurs.

3.1. The lock-free implementation using LL/SC

In [9], we formalized Herlihy's methodology [11] for transferring a sequential implementation $\mathcal{S}_a$ of any data structure into a lock-free synchronization $\mathcal{S}_{ll/sc}$ given in Fig. 2, using synchronization

Shared variable
pub: aType;
node: **array** $[1 \dots N{+}P]$ **of** nodeType;
indir: **array** $[1 \dots N]$ **of** $[1 \dots N+P]$;
Private variable
priv: bType; *pc*: $[c_1 \dots c_7]$;
x: $[1 \dots N]$; *mp*, *mi*: $[1 \dots N+P]$; *tm*, *tm1*: cType;

Program
loop
c_1: *noncrit*(pub, *priv*, *tm*, x);
loop
c_2: *mi* := *LL*(indir[x]);
c_3: *read*(node[*mp*], node[*mi*]);
c_4: **if** *guard*(node[*mp*], *priv*) **then**
c_5: *com*(node[*mp*], *priv*, *tm1*);
c_6: **if** *SC*(indir[x], *mp*) **then**
mp := *mi*; *tm* := *tm1*; **break**; **fi**;
c_7: **elseif** *VL*(indir[x]) **then break**; **fi**; **fi**;
end;
end.

Initial condition
$\Theta_{ll/sc}$: $(\forall p \in [1 \dots P] : pc.p = c_1 \ \wedge \ mp.p = N + p)$
$\wedge \ (\forall i \in [1 \dots N] : \ \texttt{indir}[i] = i)$

Liveness: weak fairness of commands $c_2, \dots, c_7$.

Fig. 2. Lock-free implementation $\mathcal{S}_{ll/sc}$ of $\mathcal{S}_a$.

```
Constant
    K ≥ N + 2P ;
Shared variable
    pub: aType;
    node: array [1 ... K] of nodeType ;
    indir: array [1 ... N] of [1 ... K] ;
    prot: array [1 ... K] of [0 ... K] ;
Private variable
    priv: bType; pc: [d10 ... d71]; suc: bool;
    x: [1 ... N]; mp, mi: [1 ... K]; tm, tm1: cType;
Program
    loop
d10:  noncrit(pub, priv, tm, x);
      loop
        loop
d20:      mi := indir[x];
d21:      ⟨ prot[mi]++; ⟩
d22:      if mi = indir[x] then break;        % goto d30
d23:      else ⟨ prot[mi]--; ⟩ fi;            % goto d20
        end;
d30:    read(node[mp], node[mi]);
d40:    if guard(node[mp], priv) then
d50:      com(node[mp], priv, tm1);
d60:      if CAS(indir[x], mi, mp) then
            tm := tm1;
d61:        ⟨ prot[mi]--; ⟩
d62:        if prot[mi] = 1 then mp := mi;
            else
d63:          ⟨ prot[mi]--; ⟩
              loop
d64:            choose mp in [1 ... K] fairly;
                if CAS(prot[mp], 0, 1) then
                  break; fi;                % goto d10
              end; fi;
            break;                          % goto d10
          else
d65:        ⟨ prot[mi]--; ⟩ fi;             % goto d20
        else
d70:      suc := (mi = indir[x]);
d71:      ⟨ prot[mi]--; ⟩
          if suc then break; fi; fi;        % goto d10 if suc, else goto d20
      end;
    end.

Initial conditions
    Θc: (∀ p ∈ [1 ... P]: pc.p = d10 ∧ mp.p = N + p) ∧ (∀i ∈ [1 ... N]: indir[i] = i)
        ∧ (∀i ∈ [1 ... K]: prot[i] = (i ≤ N+P ? 1 : 0))

Liveness: weak fairness of commands d20, ..., d71.
```

Fig. 3. Lock-free implementation $\mathcal{S}_c$ of $\mathcal{S}_a$.

primitives *LL*/*SC*. This section is only a conceptual preparation for the lock-free implementation using *CAS*, which will be presented in the Section 3.2.

Herlihy's methodology [11] can be described as follows. A process that needs access to a shared object pointed by x performs a loop of the following steps: (1) read x using an *LL* operation to gain access to the object's data area; (2) make a private copy of the indicated version of the object (note that this action need not be atomic); (3) perform the desired operation on the private copy to make a new version; (4) finally, call a *SC* operation on x to attempt to swing the pointer from the old version to the new version. The *SC* operation will fail when some other process has modified x since the *LL* operation, in which case the process has to repeat these steps until consistency is satisfied. The loop is nonblocking because at least one out of every P attempts must succeed within finite time. Of course, a process might always lose to some faster process, but this is often unlikely in practice.

In $\mathcal{S}_{ll/sc}$, we declare P extra shared nodes for private use (one for each process). Array indir acts as pointers to shared nodes, while node[$mp.p$] is taken as a "private" node of process p though it is declared publicly: other processes can read it but cannot modify it. The private variable x is intended only to determine the node under consideration, the private variable *tm* is intended to hold the result of the critical computation *com*, if executed. If some other process successfully updates a shared node while an active process p is copying the shared node to its "private" node, process p will restart the inner loop, since its private view of the node is not consistent anymore. After the assignment $mp := m$ at line *c6*, the "private" node becomes shared and the node shared previously (which contains the old version) becomes "private". Keep in mind that the composition of node and indir in $\mathcal{S}_{ll/sc}$ corresponds to Node in $\mathcal{S}_a$.

The following theorem stated in [9] is the reduction theorem that enables us to reason about a general lock-free algorithm to be designed on a high level than the synchronization primitives *LL*/*SC*. It is based on the refinement mapping. Its safety has been verified with PVS, see [9,13]. The liveness condition required is weak fairness of the commands $c_2, \ldots, c_7$. We have not formally proved that this is sufficient. The informal proof is similar to the argument in Section 4.4, but easier. In [9], we postulated strong fairness but that is stronger than necessary.

Theorem 3.1. *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is implemented by the concrete system $\mathcal{S}_{ll/sc}$ defined in Fig. 2, that is,* $\exists\, \varphi : \mathcal{S}_{ll/sc} \sqsubseteq \mathcal{S}_a$.

3.2. The lock-free implementation using CAS

We now turn our attention to the lock-free implementation using *CAS*, which is given by the algorithm $\mathcal{S}_c$ shown in Fig. 3. This lock-free implementation is inspired by the lock-free implementation $\mathcal{S}_{ll/sc}$. The lines c_2, c_6 and c_7 of $\mathcal{S}_{ll/sc}$ correspond in $\mathcal{S}_c$ to the fragments from d_{20} to d_{23}, from d_{60} to d_{65}, and from d_{70} to d_{71}, respectively.

Just as in Section 3.1, the basic idea is to implement Node[y] by node[indir[y]] for all indices y, to copy indir[$x.p$] to a private variable $mi.p$, and to try and use node[$mp.p$] as a private copy of *Node*[$x.p$]. Array prot serves to protect the indices that currently occur as values of indir[y] or $mp.p$ or $mi.p$. The loop from d_{20} to d_{23} serves to guarantee prot[$mi.p$] $\geqslant 1$ before copying node[$mi.p$] to node[$mp.p$] in d_{30}.

Command d_{30} is intended to copy the content of the shared node at indir[$x.p$] to the "private" node at $mp.p$. In the commands d_{40} and d_{50}, the abstract command a_2 is executed tentatively on the private copy node[$mp.p$]. Since the index indir[$x.p$] is guarded by prot[$x.p$], the *CAS* test in d_{60}

succeeds if and only if no other process has modified the shared node, in which case the tentative computation is committed by swapping indir[$x.p$] to $mp.p$. At this point, the "private" node becomes shared and the process decrements prot[$mi.p$] in line d_{61} since $mi.p$ no longer refers to a shared node.

When the check in line d_{62} finds that prot[$mi.p$] equals 1, this means that only this process is holding the index, and the process can thus choose this node as its "private" node by assigning the index to $mp.p$. Otherwise, the private reference $mi.p$ must be released and a new value for $mp.p$ must be chosen from the unused indices in line d_{64}. The choice is supposed to be "fair", meaning that in every infinite sequence of choices all numbers in $[1 \dots K]$ are chosen infinitely often. When an unused index $mp.p$ is chosen for private use in line d_{64}, the process increments prot[$mp.p$] to 1. Therefore, no other process will regard the chosen index as unused and take that for its private use.

In $\mathcal{S}_c$, we introduce a constant $K \geqslant N + 2P$ for the sizes of the arrays node and prot. There is a trade-off between space and time that can be tuned by the user: large K is faster when an unused index is chosen at line d_{64}, but large K requires more space.

4. Correctness

In this section, we prove that the concrete system $\mathcal{S}_c$ implements the abstract system $\mathcal{S}_a$. This result does not depend on the correctness of the lock-free implementation $\mathcal{S}_{ll/sc}$, which was proved in [9].

We introduce *NC* as the relation corresponding to command *noncrit* on aType × bType × cType and *Com* as the relation corresponding to command *com* on nodeType × bType with results in nodeType × cType. We use the abbreviation *Modif* V for $\bigwedge_{v \notin V}(v' = v)$ to denote that all variables that are not in the set V, are preserved by the transition, i.e., only variables in V can be modified. A special case is the identity relation $id = Modif\ \{\}$. We give the logical operator $\wedge$ a higher priority than $\vee$.

4.1. Invariants

We establish some invariants for the concrete system $\mathcal{S}_c$. They are clearly needed for the soundness of the design and will indeed be used in the proof of the refinement.

The first invariant *I1* expresses that the private indices *mp* of different processes differ, when the processes are not in the search loop from d_{61} to d_{64}. *I2* expresses that these indices also differ from all shared indices indir[y]. *I3* expresses that these shared indices all differ from each other.

I1: $p \neq q \ \wedge\ pc.p \notin [d_{61} \dots d_{64}] \ \wedge\ pc.q \notin [d_{61} \dots d_{64}] \ \Rightarrow\ mp.p \neq mp.q$
I2: $pc.p \notin [d_{61} \dots d_{64}] \ \Rightarrow\ \mathtt{indir}[y] \neq mp.p$
I3: $y \neq z \ \Rightarrow\ \mathtt{indir}[y] \neq \mathtt{indir}[z]$

In the expression of invariants, free variables p and q range over $[1 \dots P]$, and y and z range over $[1 \dots N]$. These three invariants thus preclude unwanted interferences. The next two invariants express that the tests in d_{60} and d_{70} correctly interpret what has happened.

$$\begin{array}{ll} I4: & pc.p = d_{60} \ \wedge \ mi.p = \mathtt{indir}[x.p] \\ & \Rightarrow \ guard(node[mi.p],\ priv.p) \\ & \wedge \ ((\mathtt{node}[mi.p],\ priv.p), (\mathtt{node}[\ mp.p],\ tm1.p)) \in Com \\ I5: & pc.p = d_{70} \ \wedge \ mi.p = \mathtt{indir}[x.p] \ \Rightarrow \ \neg guard(\mathtt{node}[mi.p],\ priv.p) \end{array}$$

The invariants *I4* and *I5* express that, at d_{60} and d_{70}, the condition $mi.p = \mathtt{indir}[x.p]$ implies that the node has been read correctly, that its value has not been changed since it was read, and that *guard*, and possibly *com*, have been computed correctly.

To prove the invariances of *I1* to *I5*, we postulate

$$\begin{array}{ll} I6: & \mathtt{prot}[i] = \sharp\{x \in [1 \ldots N] \mid \mathtt{indir}[x] = i\} \\ & +\sharp\{p \mid (pc.p \notin [d_{61} \ldots d_{64}] \ \wedge \ mp.p = i) \ \vee \ (pc.p = d_{61} \ \wedge \ mi.p = i)\} \\ & +\sharp\{p \mid (pc.p \in [d_{22} \ldots d_{71}] \ \wedge \ pc.p \neq d_{64} \ \wedge \ mi.p = i)\} \\ I7: & pc.p \in [d_{30} \ldots d_{71}] \ \wedge \ pc.p \neq d_{64} \ \wedge \ mp.q = mi.p \\ & \Rightarrow \ pc.q \in [d_{61} \ldots d_{64}] \\ I8: & pc.p \in [d_{40} \ldots d_{50}] \ \wedge \ mi.p = \mathtt{indir}[x.p] \\ & \Rightarrow \ \mathtt{node}[mi.p] = \mathtt{node}[\ mp.p] \\ I9: & pc.p = d_{50} \ \Rightarrow \ guard(\mathtt{node}[\ mp.p],\ priv.p) \end{array}$$

In *I6*, $\sharp V$ stands for the number of elements of a set V, and the free variable i ranges over $[1 \ldots K]$. Invariant *I6* precisely describes the counter $\mathtt{prot}[i]$ for all i. Notice that *I6* implies $\mathtt{prot}[i] \leqslant K$ for all i. Invariant *I7* implies that process p cannot read the "private" node of another process q. Invariant *I8* indicates that the private copy made in line d_{30} is correct as long as $mi.p = \mathtt{indir}[x.p]$ and p does not execute d_{50}. Invariant *I9* provides the precondition when process p arrives at line d_{50}.

4.2. Formalizing the algorithms

The invariants above may look reasonable, but are difficult to verify convincingly by hand. We therefore formalized the setting and used the proof assistant PVS to verify the algorithm, the above invariants, and the refinement mapping that is presented below.

For the formalization of $\mathcal{S}_a$, we specify

$$\begin{array}{l} \Sigma_0 \ = \ (\mathtt{Node}[1 \ldots N],\ \ \mathtt{pub})\,, \\ \Sigma_a \ = \ \Sigma_0 \times (\mathtt{opc}) \times (pc,\ x,\ priv,\ tm)^P\,. \end{array}$$

By this we mean that the observable state space Σ_0 is spanned by the shared variables `Node` and `pub`, that the private state spaces are spanned by the variables *pc*, *x*, *priv*, and *tm*, and that the abstract state space Σ_a is the Cartesian product of Σ_0, the space spanned by `opc`, and the private state spaces. The observation function $\Pi_a : \Sigma_a \to \Sigma_0$ removes `opc` and the private variables.

The abstract next state relation is defined by $\mathcal{N}_a = id \cup \bigcup_p \mathcal{N}_a.p$ where $\mathcal{N}_a.p = \mathcal{N}_{a_1}.p \cup \mathcal{N}_{a_2}.p$ and

$$\begin{array}{ll} (s,t) \in \mathcal{N}_{a_1}.p & \equiv \quad pc.p = a_1 \ \wedge \ pc'.p = a_2 \\ & \wedge \ Modif\{\mathtt{pub},\ priv.p,\ pc.p,\ x.p\} \\ & \wedge \ ((\mathtt{pub},\ priv.p,\ tm.p), (\mathtt{pub},\ priv.p,\ x.p)') \in NC \\ (s,t) \in \mathcal{N}_{a_2}.p & \equiv \quad pc.p = a_2 \ \wedge \ pc'.p = a_1 \ \wedge \ \mathtt{opc}' = \mathtt{opc} + 1 \ \wedge \\ & (guard(\mathtt{Node}[x.p],\ priv.p) \ \wedge \ Modif\{pc.p,\ \mathtt{Node}[x.p],\ tm.p\} \end{array}$$

$$\wedge\ ((\texttt{Node}[x.p],\ priv.p),\ (\texttt{Node}[x.p],\ tm.p)') \in Com$$
$$\vee\ \neg guard(\texttt{Node}[x.p],\ priv.p)\ \wedge\ Modif\{pc.p\}).$$

The initial conditions Θ_a and the liveness property $\mathcal{L}_a$ are given in Fig. 1.

We turn to the formalization of Fig. 3. In order to prove that the concrete system implements liveness condition $\mathcal{L}_a$ of Fig. 1, we extend the concrete code of Fig. 3 with the auxiliary variable opc which is modified (incremented with 1) at all points indicated with goto d_{10}.

We define $\Sigma_c = shared \times privstate^P$ where

$$shared = (\texttt{node}[1\ldots K],\ \texttt{indir}[1\ldots N],\ \texttt{prot}[1\ldots K],\ \texttt{pub},\texttt{opc})\ , \text{ and}$$
$$privstate = (pc,\ x,\ mp,\ mi,\ suc,\ priv,\ tm,\ tm1)\ .$$

Again, this means that these state spaces are spanned by the variables listed. The observation function $\Pi_c : \Sigma_c \to \Sigma_0$ is defined by

$$\Pi_c(s) = (s.\texttt{node} \circ s.\texttt{indir},\ s.\texttt{pub}) \in \Sigma_0$$

The projection function thus removes all private variables, as well as prot and opc, and treats node[indir[y]] as an alias of Node[y].

The concrete next state relation is defined by $\mathcal{N}_c = id \cup \bigcup_p \mathcal{N}_d.p$ where $\mathcal{N}_d.p = \bigvee_{10\leqslant i\leqslant 71} \mathcal{N}_{d_i}.p$. For the theorem prover, we needed to define all relations $\mathcal{N}_{d_i}.p$. Here, we only provide the description of concrete transitions d_{60} and d_{64}:

$$\begin{aligned}(s,t) \in \mathcal{N}_{d_{60}}.p \quad \equiv \quad & pc.p = d_{60}\ \wedge \\ & (\texttt{indir}[x.p] = mi.p\ \wedge\ pc'.p = d_{61}\ \wedge\ (\texttt{indir}[x.p])' = mp.p \\ & \ \wedge\ tm'.p = tm1.p\ \wedge\ Modif\{pc.p, \texttt{indir}[x.p], tm.p\} \\ & \vee\ \texttt{indir}[x.p] \neq mi.p\ \wedge\ pc'.p = d_{65}\ \wedge\ Modif\{pc.p\})\ .\end{aligned}$$

$$\begin{aligned}(s,t) \in \mathcal{N}_{d_{64}}.p \quad \equiv \quad & pc.p = d_{64}\ \wedge\ \exists\, k \in [1\ldots K]: \\ & (\ \texttt{prot}[k] = 0\ \wedge\ pc'.p = d_{10}\ \wedge\ (\ \texttt{prot}[k])' = 1\ \wedge\ mp'.p = k \\ & \ \wedge\ \texttt{opc}' = \texttt{opc} + 1\ \wedge\ Modif\{pc.p,\ \texttt{prot}[k],\ mp.p\} \\ & \vee\ \texttt{prot}[k] \neq 0\ \wedge\ Modif\{\}).\end{aligned}$$

This concludes the formalization of the transition system and the observation function of Fig. 3. The invariants of Section 4.1 were verified with this formalization in PVS. We postpone the treatment of the liveness property to Section 4.4.

4.3. Refinement

We now construct a refinement mapping $\varphi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$ to prove that $\mathcal{S}_c$ implements $\mathcal{S}_a$. On the state spaces, function $\varphi : \Sigma_c \to \Sigma_a$ is constructed by showing how each component of Σ_a is generated from components in Σ_c:

$$\begin{aligned}&\texttt{Node}_a[y] = \texttt{node}_c[\texttt{indir}_c[y]], \\ &pc_a.p = (pc_c.p = d_{10}\ \vee\ pc_c.p \in [d_{61}\ldots d_{64}]\ \vee\ (pc_c.p = d_{71} \wedge suc.p)? \\ &\qquad\qquad a_1 : a_2),\end{aligned}$$

where the subscript indicates the system a variable belongs to. The remaining variables of Σ_a (buf, $priv$, x, tm, opc) are identical to the variables occurring in Σ_c.

We now need to verify the four conditions for refinement mappings of Section 2.2. The verification of condition 1 follows immediately from the definitions of Π and φ. Conditions 2 and 3 have been verified with PVS. Condition 2 follows directly from the definitions of φ and Θ. For most of the transitions, the verification of condition 3 is also easy. Here, we examine in detail only transition d_{60}.

Transition d_{60} executed by process p is split into two cases according to whether $\texttt{indir}[x.p] = mi.p$ holds in the precondition.

This gives rise to the following two verification conditions:

(1) $\forall\, s,t \in \Sigma_c : \texttt{indir}[x.p] = mi.p \ \wedge\ (s,t) \in \mathcal{N}_{d_{60}}.p \ \Rightarrow\ (\varphi(s),\varphi(t)) \in \mathcal{N}_{a_2}.p.$

Using invariant $I4$, we obtain the following relation holds between the concrete states s and t:

$pc.p = d_{60} \ \wedge\ pc'.p = d_{61} \ \wedge\ guard(\texttt{node}[\texttt{indir}[x.p]],\ priv.p)$
$\wedge\ ((\texttt{node}[\texttt{indir}[x.p]],\ priv.p), (\texttt{node}[\texttt{indir}[x.p]],\ tm.p)') \in Com$
$\wedge\ Modif\{\texttt{indir}[x.p],\ pc.p,\ tm.p\}.$

This corresponds to the following relation holds between the abstract states $\varphi(s)$ and $\varphi(t)$:

$pc.p = a_2 \ \wedge\ pc'.p = a_1 \ \wedge\ guard(\texttt{Node}[x.p],\ priv.p)$
$\wedge\ ((Node[x.p],\ priv.p),\ (Node[x.p],\ tmi.p)') \in Com$
$\wedge\ Modif\{Node[x.p],\ pc.p,\ tm.p\}.$

We then conclude that $(\varphi(s),\varphi(t)) \in \mathcal{N}_{a_2}.p$.

(2) $\forall\, s,t \in \Sigma_c : \texttt{indir}[x.p] \neq mi.p \ \wedge\ (s,t) \in \mathcal{N}_{d_{60}}.p \ \Rightarrow\ (\varphi(s),\varphi(t)) \in \mathcal{N}_{a_0}.p.$

We obtain the following relation holds between the concrete states s and t:

$pc.p = d_{60} \ \wedge\ pc'.p = d_{65} \ \wedge\ Modif\{pc.p\}.$

This corresponds to the following relation holds between the abstract states $\varphi(s)$ and $\varphi(t)$:

$pc.p = a_2 \ \wedge\ pc'.p = a_2 \ \wedge\ Modif\{pc.p\}.$

We then conclude that $(\varphi(s),\varphi(t)) \in \mathcal{N}_{a_0}.p$.

4.4. Progress of the CAS implementation of the pattern

Condition 4 of Section 2.2 is verified as follows. We first prove that, analogously to the liveness condition $\mathcal{L}_a$ of Fig. 1, the system of Fig. 3 satisfies the liveness property:

$$\mathcal{L}_c : \quad \forall p \in [1\ldots P], n \in \mathbb{N} : \Box(\texttt{opc} = n \ \wedge\ pc.p \neq d_{10} \ \Rightarrow\ \Diamond(\texttt{opc} > n))\,.$$

For this purpose, we use the liveness conditions of Fig. 3 that the commands $d_{20},\ldots,d_{71}$ are treated under weak fairness and that, in every infinite sequence of choices at line d_{64}, all numbers in $[1\ldots K]$ are chosen infinitely often.

We first note that `opc`, the number of completed operations, never decreases. Therefore, if it does not increase infinitely often, it eventually remains constant. Now consider a weakly fair execution of Fig. 3 in which eventually `opc` remains constant while some processes are not at d_{10}. Because of weak fairness, these processes cycle each in one of the following three loops: the big loop d_{20}–d_{65}–d_{71}, the small loop d_{20}–d_{23}, or the small loop at d_{64}. Whenever a process p is sent back to d_{20}, the value of $\texttt{indir}[x.p]$ has been changed. This only happens when some process executes *CAS* at

d_{60} successfully. When a process executes *CAS* at d_{60} successfully, it goes to d_{61}. Since opc remains constant, this process enters and remains in the loop at d_{64}. It follows that, in this execution, all processes end up either idling in d_{10} or looping in d_{64}.

To formalize this argument, we consider the state function

$$\begin{aligned} F = {} & \sharp\{p \mid pc.p \in \{d_{21} \ldots d_{70}\} \ \wedge \ mi.p = \mathtt{indir}[x.p]\} \\ & + \sharp\{p \mid pc.p = d_{71} \ \wedge \ suc.p\} \\ & + (P-1) \cdot \sharp\{p \mid pc.p \in \{d_{61} \ldots d_{64}\}\} + (P+1) \cdot \mathtt{opc} \end{aligned}$$

Function F never decreases. In particular, notice that, when some process q modifies indir[$x.q$], the third summand increases with $P-1$ and the first summand cannot decrease with more than $P-1$. Function F increases whenever some process executes d_{20} or increments opc and goes back to d_{10}. For every process p, we also consider the state function $G.p$ given by

$$pc.p = d_j \ \Rightarrow \ G.p = (j = 20 \,?\, 72 : j)$$

Apart from the steps at d_{20} and d_{64} and the jumps back to d_{10}, function $G.p$ increases at every step of process p. It never decreases more than 61. It follows that the sum $H = 62 \cdot F + \sum_p G.p$ never decreases, and that H increases at every step apart from the looping step at d_{64}. In an execution in which opc becomes constant, function H also becomes constant since $H \leqslant A \cdot \mathtt{opc} + B$ for some constants A and B. Therefore, eventually, none of the steps d_j with $j \neq 64$ is taken anymore. Since the steps d_j with $j \neq 10$ are taken with weak fairness, it follows that, then, all steps d_j with $j \neq 10$ and $j \neq 64$ are disabled. This implies that all processes are at d_{10} and d_{64}.

Now invariant *I6* implies that there is an index i with prot[i] $= 0$, and since all processes are at d_{10} and d_{64}, this index i can be kept constant. Since the choice in d_{64} is fair, some process will eventually choose index i, succeed and increment opc. This proves liveness condition $\mathcal{L}_c$.

We finally verify condition 4 of Section 2.2 as follows. Let τ be a behavior of $\mathcal{S}_c$. We have to prove that $\varphi \circ \tau$ satisfies $\mathcal{L}_a$. Let p be a process number and $n \in \mathbb{N}$ with $pc_a.p \neq a_1$ and $\mathtt{opc} = n$ at some state in $\varphi \circ \tau$. The corresponding state of τ satisfies $pc_c.p \neq d_{10}$ and $\mathtt{opc} = n$. Now $\mathcal{L}_c$ implies that $\mathtt{opc} > n$ at some later state of τ. Therefore, also, $\mathtt{opc} > n$ at some later state of $\varphi \circ \tau$.

This concludes the proof of our reduction theorem for the lock-free implementation using *CAS*:

Theorem 4.1. *The abstract system $\mathcal{S}_a$ defined in Fig. 1 is implemented by the concrete system $\mathcal{S}_c$ defined in Fig. 3, that is, $\exists \varphi : \mathcal{S}_c \sqsubseteq \mathcal{S}_a$.*

5. Conclusions

Lock-free algorithms offer significant reliability and performance advantages over conventional lock-based implementations. Many machines provide either *CAS* or *LL/SC*, but not both. *CAS* is a weaker atomic primitive than *LL/SC* in the sense that there is a cheap and easy lock-free implementation of *CAS* with *LL/SC* while the implementations of *LL/SC* based on *CAS* either give space overhead [16], or require unbounded version numbers [6,23] for theoretical correctness.

This paper presents a general lock-free pattern based on *CAS* without giving space overhead or requiring unbounded version numbers. The lock-free pattern makes it easier to develop the lock-free implementations of any data structures. It is a *CAS* variation of Herlihy's *LL/SC* methodology

for lock-free transformation. It clearly shows that *CAS* is sufficient for practical implementations of lock-free data structures.

We present the lock-free pattern as a reduction theorem. Application of this theorem simplifies the verification effort for lock-free algorithms since fewer invariants are required and some invariants are easier to discover and easier to formulate without considering the internal structure of the final implementation. Apart from verifying the safety properties, we have also formalized the liveness property associated to lock-freedom, and informally proved that it follows from weak fairness.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our proof is not flawed, we used the higher-order interactive theorem prover PVS for mechanical support. All invariants as well as the conditions 2 and 3 of the refinement mapping φ have been verified with PVS. We felt that using PVS to prove the liveness does not give enough advantages over the handwritten proof to justify the investment and the delay in publication. We therefore defer a PVS proof of the liveness to future work. For the complete mechanical proof of safety, we refer the reader to [13].

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theoretical Computer Science 82 (2) (1991) 253–284.
[2] J.H. Anderson, M. Moir, Universal constructions for multi-object operations, in: PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 1995, pp. 184–193.
[3] R.J.R. Back, J. von Wright, Stepwise refinement of distributed systems: models, formalism, correctness: refinement calculus, in: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science, vol. 430, Springer-Verlag, 1990, pp. 42–93.
[4] G. Barnes, A method for implementing lock-free data structures, in: Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures, June 1993, pp. 261–270.
[5] B.N. Bershad, Practical considerations for non-blocking concurrent objects, in: Proceedings of the Thirteenth International Conference on Distributed Computing Systems, 1993, pp. 264–274.
[6] S. Doherty, M. Herlihy, V. Luchangco, M. Moir, Bringing practical lock-freesynchronization to 64-bit applications, in: PODC '04: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2004, pp. 31–39.
[7] H. Gao, J.F. Groote, W.H. Hesselink, Lock-free dynamic hash tables with open addressing, Distributed Computing 17 (2005) 21–42.
[8] H. Gao, J.F. Groote, W.H. Hesselink, Lock-free parallel garbage collection, in: Y. Pan, D. Chen, M. Guo, J. Cao, J. Dongarra (Eds.), Proceedings of Third International Symposium on Parallel and Distributed Processing and Applications (ISPA'05), vol. 3758, LNCS, Springer, 2005, pp. 263–274.
[9] H. Gao, W.H. Hesselink, A formal reduction for lock-free parallel algorithms, in: Proceedings of the 16th Conference on Computer Aided Verification (CAV), July 2004.
[10] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13(1) (1991) 124–149.
[11] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Transactions on Programming Languages and Systems 15 (5) (1993) 745–770.
[12] M.P. Herlihy, V. Luchangco, M. Moir, The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structure, in: Proceedings of 16th International Symposium on Distributed Computing, Springer-Verlag, October 2002, pp. 339–353.
[13] W.H. Hesselink, http://www.cs.rug.nl/wim/mechver/lockfree_reduction.

[14] W.H. Hesselink, J.F. Groote, Wait-free concurrent memory management by create, and read until deletion, Distributed Computing 14 (1) (2001) 31–39.
[15] IBM, IBM System/370 Extended Architecture, Principles of Operation, 1983.
[16] P. Jayanti, S. Petrovic, Efficient and practical constructions of LL/SC variables, in: PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2003, pp. 285–294.
[17] L. Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 872–923.
[18] V. Luchangco, M. Moir, N. Shavit, Nonblocking k-compare-single-swap, in: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, 2003, pp. 314–323.
[19] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer Verlag, 1992.
[20] H. Massalin, C. Pu, A lock-free multiprocessor OS kernel, Technical Report CUCS-005-91, Columbia University, 1991.
[21] M.M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, 2002, pp. 73–82.
[22] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, ACM Press, New York, 2002, pp. 21–30.
[23] M.M. Michael, Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS, in: DISC, 2004, pp. 144–158.
[24] M. Moir, Practical implementations of non-blocking synchronization primitives, in: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, 1997, pp. 219–228.
[25] O. Shalev, N. Shavit, Split-ordered lists: lock-free extensible hash tables, in: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, ACM Press, New York, 2003, pp. 102–111.